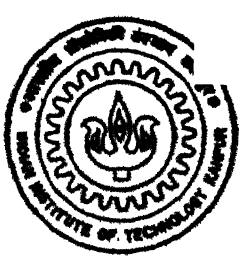
RE-ENGINEERING : COBOL TO C-SQL

by

A. Satish Kumar



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

FEBRUARY, 1996

RE-ENGINEERING: COBOL TO C-SQL

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

A. Satish Kumar

to the

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

February, 1996.

CSE-1996-M-KUM-RE

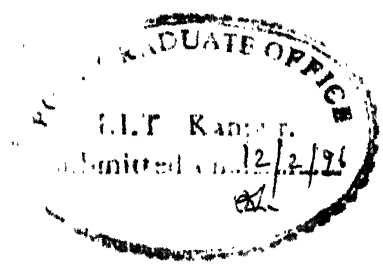
CERTIFICATE

This is to certify that the work contained in the thesis entitled **Re-engineering: COBOL to C-SQL** by **A. Satish Kumar** has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Sanjeev Kumar Aggarwal
Associate Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

Dr. T. V. Prabhakar
Associate Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

February, 1996



Acknowledgements

I am grateful to my thesis supervisors, Dr. Sanjeev Kumar Aggarwal and Dr. T. V. Prabhakar for their constant support, guidance and encouragement throughout this work.

I thank to the class of M.Tech. '94 for making my stay at IITK, a memorable period. Special thanks to my wing mates(HALL IV) Suresh Kumar, Hari Prasad, Raghu ram and Nandan babu for being wonderful company. I express my thanks to Hari Prasad and Nandan babu for proofreading this report.

Finally, I thank my parents and brother for their constant encouragement.

Abstract

The process of reengineering involves examination and alteration of an existing system in order to arrive at a new form, and a subsequent implementation of the new form. One such problem, addressed here, is migrating the existing file based systems implemented in COBOL to a relational environment. We have designed and implemented a workbench, SQLC, which automates this task to the extent possible. The subject system is re-engineered in three phases using SQLC. In the first two phases data model is recovered and converted to relational database, and in the final phase COBOL programs are restructured to C with embedded SQL. In this thesis, we shall discuss the conceptual framework used in re-engineering COBOL programs to C with embedded SQL.

Contents

Chapter 1

Introduction

1.1 Motivation

Most of the earlier information systems were implemented in COBOL. COBOL has been widely used because it has good file handling facilities. Here, the data is stored in files supported by the operating system. The main limitation of these systems is that they lack a single integrated data view. The relationship between various data items is implicitly implied by the processing actions of the programs. This results in data redundancy and difficulty in the maintenance of COBOL programs.

Data semantics, data consistency, data constraints and the relationship between various data items or data records, can be represented by a data model. The three important available data models are relational, network and hierarchical. The relational model is more successful model than the other two models.

An RDBMS environment supports a variety of functions that have to be coded explicitly in COBOL. Adhoc query support eliminates the need to write a separate program. A report generator can replace code that explicitly formats the output report. Other useful features include security, data integrity, roll back, crash recovery etc.

The maintenance of the information systems implemented in COBOL consumes most of the MIS resources. But they are still being continued as a lot of money has already been invested in them. It is becoming increasingly desirable to migrate these information systems to a relational platform, so

that the features available in an RDBMS environment can be extended to it and maintenance costs can be cut down. After migration, as all the file access statements are handled by RDBMS, there is no need to stick to COBOL. C language would be better as it is quite popular. The task of migration of a subject system to an RDBMS environment and the translation of a source program comes under the purview of re-engineering.

1.2 Problem definition

The aim of this project is to develop a workbench which provides an environment to re-engineer existing file based systems implemented in COBOL to an Ingres relational platform. This involves recovering data model of the subject system and converting it to a relational database, and transforming the COBOL programs into C with embedded SQL.

1.3 Related work

This problem has attracted wide attention. We list some of the tools developed in this regard.

- Refine is a workbench that supports following features in working with COBOL legacy systems:
  - Aids in understanding code structure.
  - Generates documentation.
  - Aids in assessing and improving code quality.
- P. L. Srinivas at IIT Kanpur developed COBSQL [1], a tool that transforms the COBOL programs into programs which accesses a relational database. This involves replacing all file access statements by equivalent SQL statements.
- Lately, work has been going on, on a project COBCY for the development of a compiler that transforms COBOL programs into C programs [2]. But, it is still in an embryonic stage.

1.4 Work done

In the present work we have removed some of the limitations of COBSQL and implemented a workbench named SQLC that migrates COBOL programs to an RDBMS environment. SQLC migrates COBOL programs to the target environment in three phases - Parsing, Unification and Conversion.

- *Parsing*: A grammar based tool is used to extract the data view as seen by each program [3].
- *Unification*: A tool named Unifier is used to unify the the extracted data views to a single integrated data view. The recovered data model is represented in an intermediate form which is used in a later phase [3].
- *Conversion*: A grammar based tool named Converter is used to restructure COBOL programs into C with embedded SQL. Converter takes a COBOL program and the recovered data model as input and outputs C with embedded SQL.

1.5 Organization of the thesis

The rest of the report is organized as follows

- **Chapter 2** outlines the main features of COBOL.
- **Chapter 3** presents an overview of data management in the target environment.
- **Chapter 4** outlines various phases in migration of COBOL programs to RDBMS environment and also describes the various rules used in restructuring COBOL programs to C with embedded SQL.
- **Chapter 5** presents an overview of the various tools provided by SQLC.
- **Chapter 6** evaluates the characteristics of the re-engineered system and also outlines the test suite followed to test the restructuring rules.

- **Chapter 7** summarizes the work done and suggests extensions to the project.

Chapter 2

Source environment: COBOL and flat file

In this chapter we outline the main features of COBOL which are relevant to our present work (Phillipakis and Kajmier [4], Dastidar [5]).

2.1 Data types

In this section we describe briefly the main data types supported by COBOL.

2.1.1 Elementary data types

The elementary data types supported by COBOL are numeric, alphanumeric, alphabetic, numeric edited and alphanumeric edited. They are discussed briefly here.

- *Numeric*:A numeric data item is used to store a number. A data item is said to be of numeric type, if its picture class description has only the following symbols - 9, P, S and V.
- *Alphabetic*: An alphabetic data item is used to store characters. A data item is said to be of alphabetic type, if its picture class description has only the following symbols - A and B.

- *Alphanumeric*: An alphanumeric data item is used to store a number or characters. A data item is said to be of alphanumeric type, if its picture class description has only the following symbols - 9, X and A.

- *Numeric edited*: A numeric edited data item is used to store a number. When data is moved to this data item, the value gets edited according to the specification made in its picture class description. A data item is said to be of numeric edited type, if its its picture class description has the following symbols - B, /, Z, 0, +, -, *, ,, ., CR, DB, and $.

- *Alphanumeric edited*: An alphanumeric data item is used to store a number or characters. When data is moved to this data item, the value gets edited according to the specification made in its picture class description. A data item is said to be an alphanumeric edited type, if its picture class description has the following symbols - X, 9, A, B, 0, and /.

Condition name

A Condition name is associated with a variable and specifies either a single or a set of values for the same.

For example,

```
77 DEGREE-TYPE.
   88 B-TECH VALUE IS ONE.
   88 M-TECH VALUE IS TWO.
```

In the above example, B-TECH and M-TECH are Condition names associated with the variable DEGREE-TYPE.

2.1.2 Group item

A group item is made up of fields which are either elementary or again group items, resulting in a tree like structure. This structure is described in COBOL using level numbers.

For example

```
01 CARD.
```

02 NAME PICTURE IS X(25).
02 STREET PICTURE IS X(25).
02 CITY PICTURE IS X(30).

2.1.3 Table

A table is a list of similar data items. It eliminates the need for separate entries for repeated data items.

For example,
01 SALES OCCURS 4 TIMES PICTURE IS 99999.

2.1.4 File

In COBOL, a file is treated as a sequence of records. COBOL supports three types of file organizations namely sequential, relative and indexed. A file can be accessed in one of the three modes - sequential, random or dynamic. In dynamic mode, a file can be accessed either sequentially or randomly.

In the sequential file organization, records are stored in the same order they are written in. Also, records can be accessed only in the order in which they are stored in the file.

In the relative file organization, records are identified by a relative record number. The relative record number specifies the location of the record from the beginning of the file. This file can be accessed in three different modes namely sequential, random or dynamic. For the random mode, one has to specify the relative record number.

In the indexed sequential file organization, the file is organized on its primary key field. Records are stored in either ascending or descending order of the key. This file too can be accessed in any of the three different modes mentioned before. For the random mode, one has to specify value of the key.

2.2 Program structure

Every COBOL program consists of a set of instructions. This set is divided into four divisions namely Identification division, Environment division, Data division and Procedure division.

Identification division is the first division of every COBOL program and is used for documentation purposes. In our context this division is relatively unimportant.

Environment division specifies details of the computer and all the peripherals used by the program. It has two sections namely Configuration section and Input-output section. The Configuration section contains an overall specification of the computer used for the purpose of compilation and execution of the program. The Input-output section specifies all the files used by the program and the type of hardware required for each file.

Data division gives a detailed description of every data item used in the program. It has four sections namely File section, Working Storage section, Linkage section, and Report section. File section provides a detailed specification of the structure for each file used in the program. Working storage section is used to describe the data items which hold intermediate results in processing. Linkage section specifies declarations of the data items used in communicating with subroutines. Report section is used to describe the data items used for generating reports.

Procedure division consists of statements which specify processing actions on the data items declared in the Data division.

2.3 Control flow

Procedure division of a COBOL program is composed of one or more paragraphs. A paragraph is the basic unit of processing. Under the rules of COBOL, a paragraph can be executed in any of the three ways as described below:

1. By Perform statement.

   For example,
   PERFORM READING-RECORDS.

   This statement causes control to be transferred to the paragraph READING-RECORDS. After the execution of paragraph READING-RECORDS, control returns to the statement lexically following the Perform statement.

2. By Goto statement.

   For example,
   GO TO ERROR-PARA.

   This statement transfers control to the paragraph ERROR-PARA, with no return at the end of that paragraph.

3. A paragraph can be executed as a result of fall thru. This will result, when the execution of a paragraph reaches the end of it and has not been called by Perform statement. At this point control passes to the paragraph which lexically follows the current paragraph.

2.4 Subroutine

In COBOL a subroutine has USING phrase in the header of Procedure division. The general syntax for Procedure division of subroutine is -

PROCEDURE DIVISION USING data-name-1 [,data-name-2] ...

The data-name-1, data-name-2 etc are described in the Linkage section of the program. A subroutine can be executed by a Call statement. The Call statement transfers control from the calling program to the subroutine. It also copies the value of data items in the Call statement to the corresponding data items in the procedure division header of the subroutine. After execution of the subroutine, values of the data items specified in its USING phrase are copied back to the corresponding data items specified in the Call statement of the calling program.

Chapter 3

Target environment: Relational database and SQL

In this chapter we present an overview of data management in the target environment [6].

3.1 Relational database

In a relational database relationships among the data are represented by tables. A table is a two dimensional structure with rows and columns . A row in the table represents a relationship between the set of values. Each row in the table is uniquely identified by the value of the primary key.

3.2 Query languages

Query languages are used to access data from the relational database. These languages are at a higher level than application programming languages like C. The three important query languages available are SQL, Quel and QBE. SQL is the most widely used query language.

3.2.1 SQL

The SQL provides following features:

- *Data definition language*: The SQL data definition language provides commands for defining tables and views, deleting tables and modifying tables.

- *Data manipulation language*: The SQL data manipulation language provides commands to manipulate data stored in the relational database.

- *Embedded SQL statements*: Application programs can access the data stored in the relational database through SQL statements embedded in it.

- *Integrity and Security*: The SQL includes commands to specify begin and end of transactions. It also provides commands to specify integrity constraints and integrity checking on data stored in the relational database.

3.3 C application program interface

A C program can access the relational database through embedded SQL statements. These commands are used to declare host variables, to define cursor, to manipulate cursor and table and to include code for error handling.

3.3.1 Host variable

In C with embedded SQL, data transfer between an object program and a database is done through host variables. Host variables can be of three types namely integer, float and char.

3.3.2 Data manipulation

Embedded SQL provides the following SQL commands to manipulate data stored in the table.

- Select, returns one or more rows from the specified table.

- Insert, adds new rows into the specified table.

- Delete, deletes one or more rows from the specified table.

- Update, changes the data in one or more columns of the specified table.

3.3.3 Cursor

A cursor is a pointer to individual rows in a SQL query result. Embedded SQL supports the following SQL commands to manipulate the cursor.

- Declare, used to define the cursor.
- Open, used to determine the set of rows pointed by the cursor and allocate memory for them.
- Fetch, copies the contents of a row into the specified host variables and advances the position of the opened cursor to the next row.
- Update, used to update values of the current row of the specified cursor.
- Close, used to free the memory allocated to handle the cursor.

Chapter 4

Re-engineering from COBOL to SQL and C

This chapter outlines the various phases in re-engineering and enumerates the rules used to restructure COBOL programs to C with embedded.

4.1 Phases in Re-engineering

The existing systems implemented in COBOL are re-engineered to a relational environment in three phases namely Parsing, Unification and Conversion. In the first two phases, we re-engineer the data model. The subject system does not have an integrated view of data because the data view as seen by each program may be different. The various conflicts that can occur between the different data views of programs are listed below

1. The same data item has different structures in different programs. This is because the programmer describes only those data items which he wants to refer and declares the rest of them as FILLER.

2. The same file or record or data item is declared by different names in different programs. This is due to not following a uniform naming convention.

3. The same data item is defined by different types in different programs.

So, the methodology adopted in re-engineering data model is to extract the data view as seen by each program and unify these views by an unifying algorithm [3]. The unified view is represented in an intermediate form named Unydef which is used in a later phase. Unydef provides correspondence information between the record fields of the file and the fields of the table. The Unydef is explained below by an example.

The meanings for the various symbols used in representing the data model are:

s: Field type is char
k: Key field
n: Not a key field
c: Name for the field is changed
u: Name for the field is unchanged
r: Record field is unified to more than one table field.
In further discussion, we refer to these table fields as *sub- fields.*

The data view as seen by program A for the file CARD-REC is:

CARD_NAME s 25 k,
CARD_HOUSE_NO s 10 n,
FILLER_1 s 55 n

The data view as seen by program B for the file CARD-REC is:

CARD_NAME s 25 k,
FILLER_1 s 10 n,
CARD_STREET s 25 n,
CARD_CITY s 30 n

After unification, the recovered relation for the file CARD-REC is:

CARD_NAME s 25 k,
CARD_HOUSE_NO s 10 n,
CARD_STREET s 25 n,
CARD_CITY s 30 n,

Then Unydef for the file CARD-REC in program A is:

CARD_NAME s 25 u,
CARD_HOUSE_NO s 10 u,

```
FILLER_1 r {
        CARD_STREET s 25 u,
        CARD_CITY s 30 u
}
```

Unydef for the file CARD-REC in program B is:

```
CARD_NAME s 25 u,
CARD_HOUSE_NO s 10 u,
CARD_STREET s 25 u,
FILLER_1 s 30 c CARD_CITY
```

In the final phase, using restructuring rules we transform COBOL programs into C with embedded.

4.2 Restructuring rules

In this section we enumerate restructuring rules for the main features of COBOL.

4.2.1 Data names

In COBOL names of data item are formed only with following characters: 0-9,A-Z and -.

The conversion rule for data names is:
Replace the character "-" with character "_". COBOL data names are not case sensitive. We take this feature as advantage in forming names for new variables which are used to simulate semantics of COBOL.

4.2.2 Data types

Elementary data types

The elementary data types supported by COBOL are numeric, alphanumeric, alphabetic, numeric edited, alphanumeric edited.

Data items of numeric type are converted to either integer or float type.
For example,

```
  01 NO-OF-RECORDS PICTURE IS 999.
  01 MARKS PICTURE IS 999V99.
```
The above declarations are replaced with
```
  int NO_OF_RECORDS;
  float MARKS;
```

Data items of alphanumeric, alphabetic, numeric edited, alphanumeric edited type are converted to char type.
For example,
```
  01 NAME PICTURE IS X(25).
```
The above declaration is replaced with
```
  char NAME[25 + 1];
```

Condition name

A Condition name is converted to a function which checks whether the variable to which it is associated is within the specified range.

For example,
```
 77 DEGREE-TYPE.
 88 M-TECH VALUE IS ONE.
```
The above declaration for the Condition name M-TECH is replaced with
```
 int M_TECH() {
     if( DEGREE_TYPE == ONE)
        return TRUE;
     else
        return FALSE;
 }
```

Group item

The group items declared in Working Storage section are converted to structures.

For example,
```
  01 BALANCE-DATA.
     02 ITEM-NUMBER PICTURE IS 99999.
     02 PART-NUMBER PICTURE IS X(14).
```

```
    02 ITEM-NAME PICTURE IS 99999.
```
The above declaration is replaced with
```
  struct {
    int ITEM_NUMBER;
    char PART_NUMBER[14 + 1];
    int ITEM_NAME;
  }BALANCE_DATA;
```

Table

Table data items are converted to arrays.

For example,
```
  01 SALES OCCURS 4 TIMES PICTURE IS 99999.
```
The above declaration is replaced with
```
  int sales[4];
```

File

A file is converted to a table. In COBOL, data transfer between files and object program is done through records described in the File division. In C with embedded SQL, it is done through host variables. Record fields declared in the File section are replaced with host variables. The name of the host variable is formed by concatenating the names of all top level data items to the name of the corresponding record field. This is necessary, because in COBOL the name of the record field need not be unique as record structure is tree like and can be qualified by top level data items. But in relational database the structure of the table is flat. This means name for every host variable must be unique.

For example,
```
  01 CARD.
    02 NAME PICTURE IS X(25).
    02 STREET PICTURE IS X(25).
    02 CITY PICTURE IS X(30).
```
The above declarations are replaced with
```
  EXEC SQL begin declare section;
    char CARD_NAME[25 + 1];
```

```
    char CARD_STREET[25 + 1];
    char CARD_CITY[30 + 1];
  EXEC SQL end declare section;
```

Unification results in a single integrated view of data. Due to this a record field can be unified to more than one field of the table. If this happens, we have to declare host variables for *sub-fields*. This may result in name clashes while declaring host variables. One can avoid the name clash problem by concatenating *sub-field* name with SqlC_#NO, where #NO is unique for every field.

For example,
The file description for the file CARD-REC is

```
  01 CARD.
     02 ADDRESS PICTURE IS X(80).
```

Let Unydef for the file CARD-REC is

```
  ADDRESS r {
     CARD_NAME s 25,
     CARD_ STREET s 25,
     CARD_CITY s 30,
  }
```

Then host variable declarations for the file CARD-REC are

```
  EXEC SQL begin declare section;
     char CARD_ADDRESS_1[80 + 1];
     char CARD_NAMESqlC1[25 + 1];
     char CARD_STREETSqlC2[25 + 1];
     char CARD_CITYSqlC3[30 + 1];
  EXEC SQL end declare section;
```

4.2.3 Arithmetic operators

There are five binary arithmetic operators in COBOL. Operators +, -, * and / mean addition, subtraction, multiplication and division respectively. These operators need not be replaced as they are supported by C language. Operator ** stands for exponential. This operator is replaced with pow(...) a library function, as it is not available in C language.

For example, 2**3 is replaced with pow(2,3);

4.2.4 Conditions

In COBOL conditions are categorised as Relation condition, Sign condition, Class condition, Condition name condition, Negated simple condition and Compound condition.

Relation condition

The Relation condition specifies comparison of two operands.

The conversion rules for the Relation condition are:

- *Format 1*:

  operand-1 IS {
  GREATER THAN | >
  | LESS THAN | <
  | EQUAL TO | =
  } operand-2

  If operands are of numeric type then the conversion rule for the above format is

  operand-1 {
  | > | >
  | < | <
  | == | ==
  } operand-2

  If operands are of non-numeric type then the conversion rule for the above format is

  {
  strcmp(operand-1, operand-2) > 0 |
  strcmp(operand-1, operand-2) > 0 |
  strcmp(operand-1, operand-2) < 0 |
  strcmp(operand-1, operand-2) < 0 |
  strcmp(operand-1, operand-2) == 0 |
  strcmp(operand-1, operand-2) == 0 |
  }

- *Format 2*:

```
operand-1 IS NOT {
    | GREATER THAN | >
    | LESS THAN | <
    | EQUAL TO | =
} operand-2
```

If operands are of numeric type then the conversion rule for the above format is

```
operand-1 {
    | <= | <=
    | >= | >=
    | != | !=
} operand-2
```

If operands are of non-numeric type then the conversion rule for the above format is

```
{
    strcmp(operand-1, operand-2) <= 0 |
    strcmp(operand-1, operand-2) <= 0 |
    strcmp(operand-1, operand-2) >= 0 |
    strcmp(operand-1, operand-2) >= |
    strcmp(operand-1, operand-2) != 0 |
    strcmp(operand-1, operand-2) != 0 |
}
```

Sign condition

The Sign condition evaluates whether the value of an arithmetic expression is less than, greater than or equal to zero.

The conversion rules for the Sign condition are:

- *Format 1*:
  arithmetic-expression IS {POSITIVE | NEGATIVE | ZERO }
  is replaced with
  arithmetic-expression { > 0 | < 0 | == 0 }

- *Format 2*:
  arithmetic-expression IS

NOT {POSITIVE | NEGATIVE | ZERO }

is replaced with

arithmetic-expression { <= 0 | >= 0 | != 0 }

Class condition

The Class condition determines the type of operand.

The conversion rules for the Class condition are:

- *Format 1*:

  operand IS [NOT] ALPHABETIC

  is replaced with

  [!]IsAlphaSqlC(operand)

  Where IsAlphsSqlC() is a library function which tests whether operand is of alphabetic type.

- *Format 2*:

  operand IS [NOT] NUMERIC

  is replaced with

  [!]IsNumericSqlC(operand)

  Where IsNumericSqlC() is a library function which tests whether operand is of numeric type.

Condition name condition

The Condition name condition evaluates whether the variable to which Condition name is associated, is within the specified range.

For example,

```
77 STUDENT-DEGREE.
   88 BTECH VALUE IS ONE.
   88 MTECH VALUE IS TWO.
   ...
IF BTECH THEN ...
ELSE ...
```

The Condition name condition in IF statement is replaced with function call BTECH(). The function BTECH() returns boolean value true or false depending up on the value of the variable STUDENT-DEGREE.

Negated simple condition

A Simple condition can be negated by a NOT operator.

The conversion rule for the Simple condition is:
NOT simple-condition
is replaced with
! simple-condition

Compound condition

A Compound condition is formed by connecting conditions with logical operators. The only logical operators available in COBOL are AND and OR.

The conversion rule for the Compound condition is:
condition-1 {AND | OR } condition-2 ...
is replaced with
condition-1 { && | || } condition-2 ...

4.2.5 Move statement

The Move statement is used to copy data from one data item to another data item. The receiving or sending operand of Move statement can be either an elementary data item or a group data item. When both the operands are elementary data items then the data movement is called Elementary move else it is called Group move.

The conversion rule for the Elementary Move statement is:
The Move statement is replaced either by an assignment statement or by a string copy statement or by a library function depending on the type of the sending and receiving fields.

For example,
MOVE A TO B.
If both A and B are of alphanumeric type then the above
MOVE statement is replaced with
strncpy(B, A, operand-1);
where operand-1 is size of the data item B.

If A is of numeric type and B is of alphanumeric type then
the above MOVE statement is replaced with

MoveSqlC("format of A", A, "format of B", B);

where MoveSqlC is a library function.

If both A and B are of numeric type then the above statement
is replaced with

B = A;

COBOL supports integer and float types with variables sizes, while C supports these with predefined sizes. In COBOL, when a value larger than size of data item is moved to it then truncation occurs. The above semantics of COBOL can be simulated in C by providing a library function which truncates the value properly, if needed.

When data is moved to editable data item, it gets edited according to description specified in its picture class. In C we can simulate the same by calling a library function.

The conversion rule for the Group Move statement is:
The Group Move statement is replaced with following statements

- Contents of the sending fields of the Move statement are copied in to temporary variable tempSqlC;
- Data is moved to the receiving fields of the Move statement from tempSqlC;

4.2.6 Arithmetic statements

The arithmetic statements that are supported by COBOL are Add statement, Subtract statement, Divide statement, Multiply statement and Compute statement.

Add statement

The Add statement is used to add one or more numeric operands and store the sum.

The conversion rules for the Add statement are:

- *Format 1*:

  ADD operand-1 [,operand-2] ... TO operand-3
  [,operand-4 *]* ... .

  is replaced with

  operand-3 += operand-1 *[*+ operand-2 *]* ...;
  *[*operand-4 += operand-1 *[*+ operand-2 *]* ...; *]*
  ...

- *Format 2*:

  ADD operand-1 *[*,operand-2*]* ... GIVING operand-3
  [,operand-4*]* ... .

  is replaced with

  operand-3 = operand-1 *[*+ operand-2*]* ...;
  *[*operand-4 = operand-1 *[*+ operand-2*]* ...; *]*
  ...

Subtract statement

The Subtract statement is used to subtract one or sum of two or more numeric operands from one or more numeric operands and store the result.

The conversion rules for the Subtract statement are:

- *Format 1*:

  SUBTRACT operand-1 *[*,operand-2*]* ... FROM operand-3
  [,operand-4*]* ... .

  is replaced with

  operand-3 -= operand-1 *[*+ operand-2*]* ...;
  *[*operand-4 -= operand-1 *[*+ operand-2 *]* ...; *]*
  ...

- *Format 2*:

  SUBTRACT operand-1 *[*,operand-2 *]* ... FROM operand-3
  [,operand-4 *]* ... GIVING operand-5 *[* ,operand-6 *]* ...

  is replaced with

  operand-5 = operand-1 *[*+ operand-2 *]* ...-operand-3
  [-operand-4 *]* ...;
  *[*operand-6 = operand-1 *[*+ operand-2*]* ... -operand-3
  [-operand-4*]* ...; *]*

...

Multiply statement

The Multiply statement is used to multiply a numeric operand by another and store the result.

The conversion rules for the Multiply statement are:

- *Format 1*:
  MULTIPLY operand-1 BY operand-2 *[*,operand-3 *]* ....
  is replaced with
  operand 2 * operand 1;
  *[*operand 3 * operand-1; *]*
  ...

- *Format 2*:
  MULTIPLY operand-1 BY operand-2 *[*,operand-3*]* ...
  GIVING operand-4 *[*, operand-5*]* ... .
  is replaced with
  operand-4 = operand-2 * operand-1;
  *[*operand 5 = operand-3 * operand-1; *]*
  ...

Divide statement

The Divide statement is used to divide one or more numeric operands by a numeric operand and store the result.

The conversion rules for the Divide statement are:

- *Format 1*:
  DIVIDE operand-1 INTO operand-2 *[*,operand-3 *]* ... .
  is replaced with
  operand-2 /= operand-1;
  *[*operand-3 /= operand-1; *]*
  ...

- *Format 2*:
  DIVIDE operand-1 INTO operand-2 *[*,operand-3 *]* ...
  GIVING operand-4 *[*, operand-5*]* ... .

is replaced with
operand-4 = operand-2 / operand-1;
[operand-5 = operand-3 / operand-1;]
...

- *Format 3*:
DIVIDE operand-1 BY operand-2 GIVING operand-3
[,operand-4] ... .
is replaced with
operand-3 = operand-1 / operand-2;
[operand-4 = operand-1 / operand-2;]
...

- *Format 4*:
DIVIDE operand-1 BY operand-2 GIVING operand-3
REMAINDER operand-4
is replaced with
operand-3 = operand-1 / operand-2;
operand-4 = operand-1 % operand-2;

Compute statement

The Compute statement is used to assign to one or more numeric operands the value of arithmetic expression.

The conversion rule for the Compute statement is:

COMPUTE operand-1 [,operand-2] ... = arithmetic-expression.
is replaced with
operand-1 = arithmetic-expression;
[operand-2 = arithmetic-expression;]
...

4.2.7 Sequence control statements

These statements are used to alter the flow of control.

Perform statement

The Perform statement is used to transfer control to one or more paragraphs and return the control to current paragraph after their execution. Perform statement is replaced either by a function call or by a Goto statement depending on whether the paragraph is converted to a function or a block. In a typical COBOL program a paragraph can be executed by any of the three ways mentioned earlier. A paragraph cannot be converted into a function, if the COBOL program contains both Goto and Perform statements. Because in such programs we can replace the Perform statement by a function call, but there is no equivalent statement for the Goto statement. One solution to this problem is as follows.

If the COBOL program does not have a Goto statement then we convert the paragraph to a function. If the COBOL program contains both Goto and Perform statements then we convert the paragraph to a block.

The conversion rules for the Perform statement when it is replaced with a function call are:

- *Format 1*:
  PERFORM para name 1 THRU para-name-2
  is replaced with
  ```
  para name 1();
        ...
  para name 2();
  ```

- *Format 2*:
  PERFORM para name 1 THRU para-name-2 operand TIMES
  is replaced with
  ```
  { int i = (operand > 0)?operand:0;
        while(i- -) {
            para-name-1();
            ...
            para-name-2();
        }
  }
  ```

- *Format 3*:
  PERFORM para-name-1 THRU para-name-2

```
        UNTIL condition.
is replaced with
 while(! condition) {
        para-name-1();
        ...
        para-name-2();
 }
```

- *Format 4*:

```
PERFORM para-name-1 THRU para-name-2 VARYING
        operand-1 FROM operand-2 BY operand-3
            UNTIL condition-1
        [AFTER operand-4 FROM operand-5 BY operand-6
            UNTIL condition-2]
        [AFTER operand-7 FROM operand-8 BY operand-9
            UNTIL condition-3]
is replaced with
  for(operand-1 = operand-2; !condition-1;
            operand-1+= operand-3)
        [for(operand-4 = operand-5; !condition-2;
            operand-4+= operand-6) ]
        [for(operand-7 = operand-8; !condition-3;
            operand-7+= operand-9)] {
            para-name-1();
            ...
            para-name-2();
  }
```

The conversion rule for the Perform statement when it is replaced with a Goto statement is:

The Perform statement returns control to the current paragraph implicitly after executing the specified paragraph. One can simulate the semantics of the Perform statement in C by following the rules specified below.

- For every Perform statement generate a new label and a number. Generated number is used to identify label.

- Replace the Perform statement by a function call to push the generated label number into stack and the Goto statement to the block corresponding to the paragraph specified in it.
- Place the generated label next to the Goto statement.
- At the end of each block place a function call, which checks whether control is transferred to that block by Perform statement, and transfers control to the Perform handler if it is so.
- Place the Perform handler at the end of program which transfer controls to the statement next to label by examining the contents of the top element of the stack.

```
PushSqlC(label-no);
goto para-name;
SqlC_label-no: ;
...

para-name: {
    ...
    if(IsPrfrmStmtSqlC())
        goto PrfrmHndlSql;
}
    ...
    /*Perform handler*/
PrfrmHndlSqlC :{
    switch(PopSqlC()) {
        case label-no:
        goto SqlC_label-no;
        ...
    }
}
```

PushSqlC: This function Pushes the label-no into stack.
IsPrfrmStmt: This function checks whether control is transferred to the paragraph by the Perform statement.
PopSqlC: This function pops the top element of the stack.

Goto statement

The Goto statement is used to transfer control from one paragraph to another.

The conversion rule for the Goto statement is:

- *Format 1*:
  GOTO para-name.
  is replaced with
  goto para-name;

- *Format 2*:
  GOTO para name 1 *[*,para-name-2*]* ...
  DEPENDING ON operand-1.
  is replaced with
  switch(operand-1) {
  case 1:
  goto para-name-1;
  *[*case 2:
  goto para-name-2;
  ]
  ...
  }

If statement

The If statement evaluates the condition and the subsequent actions of object program depends up on the value of the condition.

The conversion rules for the If statement are:

- *Format 1*:
  IF condition THEN statements.
  is replaced with
  if(condition) {
  statements;
  }

- *Format 2*:
  IF condition THEN statements-1

```
        ELSE statements-2.
is replaced with
  if(condition) {
        statements-1
  }
  else {
        statements-2
  }
```

Stop statement

The Stop statement is used to terminate the execution of the object program.

The conversion rule for the Stop statement is:

```
  STOP RUN.
is replaced with
  exit(0);
```

Call statement

The Call statement is used to transfer control from one object program to another.

The conversion rule for Call statement is:

```
  CALL prog-name USING operand-1 [,operand-2 ] ...
is replaced with
  prog name(parameter-1 [,parameter-2] ... );
```

Where parameter 1 is address of the operand-1.

Exit Program statement

The Exit Program statement is used to terminate the execution of the object program and return the control to the calling program.

The conversion rule for the Exit program statement is:

```
  EXIT PROGRAM
is replaced with
  return;
```

4.2.8 Input-Output statements

The Input-output statements are used to read or write the data items from input or output devices.

Accept statement

The Accept statement is used to scan data for the specified data items.
The conversion rules for the Accept statement are:

- *Format 1*:
  ACCEPT operand-1.
  is replaced with
  scanf("format", parameter-1);
  Where parameter-1 is the address for operand-1.

- *Format 2*:
  ACCEPT operand-1 FROM DATE.
  is replaced with
  DataSqlC(tempSqlC);
  sscanf(tempSqlC, "format", parameter-1);
  Where parameter-1 is the address for operand-1 and DataSqlC is a library function to get the current date.

- *Format 3*:
  ACCEPT operand 1 FROM DAY.
  is replaced with
  DaySqlC(tempSqlC);
  sscanf(tempSqlC, "format", parameter-1);
  Where parameter-1 is the address for operand-1 and DaySqlC is a library function to get the current day.

- *Format 4*:
  ACCEPT operand-1 FROM TIME.
  is replaced with
  TimeSqlC(tempSqlC);
  sscanf(tempSqlC, "format", parameter-1);
  Where parameter-1 is the address for operand-1 and DataSqlC is a library function to get the current time.

Display statement

The Display statement is used to display the specified data items on the console.

The conversion rule for the Display statement is:
DISPLAY operand-1 *[*,operand-2*]* ...
is replaced with
printf("format", operand-1 *[*,operand-2*]* ...);

4.2.9 File manipulation statements

In COBOL, a file can be accessed in any of three modes - sequential, random or dynamic. If a file is accessed sequentially, then we can simulate it in C with embedded SQL by declaring cursor for the table corresponding to the file. Otherwise, we have to use the SQL query with key as the selection criterion. A file with relative organization is handled by including a field in the corresponding table that gives relative record number. The following sections list the restructuring rules for the main file access statements available in COBOL. For the file descriptions and table definitions refer to Appendix A.

Open statement

The Open statement is used to initiate file processing. The Open statement is replaced with the following statements, only if the file specified in the Open statement is accessed sequentially and is opened in the input mode.

- Cursor definition for the table corresponding to file.
- SQL statement to open the cursor.

For example,
OPEN INPUT CARDS-IN.
is replaced with
EXEC SQL declare CARD_IN
cursor for
select

```
                card name,
                card_street,
                card_city
            from card in;
  EXEC SQL open CARDS_IN;
```

Close statement

The Close statement is used to terminate the file processing. The Close statement is replaced with SQL Close statement, only if cursor is defined for the table corresponding to file specified in the Open statement.

For example
```
  CLOSE CARDS IN.
```
is replaced with
```
  EXEC SQL close CARDS IN;
```

Read statement

The Read statement is used to read a record from the file. If cursor is defined for the table corresponding to the file specified in the Read statement then the Read statement is replaced with an SQL query, which reads a tuple from the cursor.

For example,
```
  READ CARD IN.
```
is replaced with
```
  EXEC SQL fetch CARD_IN into
          :CARD_NAME,
          :CARD_STREET,
          :CARD_CITY
          ;
```

If cursor is not defined for the table corresponding to the file specified in the Read statement then the Read statement is replaced with an SQL query, which reads a tuple from the table.

For example,

```
  READ CARD_IN.
```

is replaced with

```
  EXEC SQL
     select
       card_name,
       card_street,
       card_city
     into
       :CARD_NAME,
       :CARD_STREET,
       :CARD_CITY
     from card_in
     where card_name = :CARD_NAME;
```

Write statement

The Write statement is used to write a record into a file. The Write statement is replaced with an SQL query, which inserts a tuple into the table corresponding to the file.

For example,

```
  WRITE CARD.
```

is replaced with

```
  EXEC SQL insert into card_in
     (
     card_name,
     card_street,
     card_city
     )
     values (
     :CARD_NAME,
     :CARD_STREET,
     :CARD_CITY
     );
```

Rewrite statement

The Rewrite statement is used to replace a record in the specified file.

If cursor is defined for the table corresponding to the file specified in the Rewrite statement then the Rewrite statement is replaced with a SQL query, which updates the current row pointed by cursor.

For example,

```
  REWRITE CARD_IN.
is replaced with
  EXEC SQL update card_in
     set
       card_name = :CARD_NAME,
       card_street = :CARD_STREET,
       card_city = :CARD_CITY
     where current of CARD_IN;
```

If cursor is not defined for the table corresponding to the file specified in the Rewrite statement then the Rewrite statement is replaced with an SQL query, which updates a tuple in the table.

For example,

```
  REWRITE CARD_IN.
is replaced with
  EXEC SQL update card_in
     set
       card_name = :CARD_NAME,
       card_street = :CARD_STREET,
       card_city = :CARD_CITY
     where card_name = :CARD_NAME;
```

Delete statement

The Delete statement is used to delete a record from the specified file.

If cursor is defined for the table corresponding to the file specified in the Delete statement then the Delete statement is replaced with an SQL query, which deletes current row pointed by the cursor.

For example,

```
  DELETE CARD_IN.
```

is replaced with

```
  EXEC SQL delete from card_in
          where current of CARD_IN;
```

If cursor is not defined for the table corresponding to the file specified in the Delete statement then the Delete statement is replaced with a SQL query, which deletes a tuple from the table.

For example,

```
  DELETE CARD_IN.
```

is replaced with

```
  EXEC SQL delete from card_in
     where card_in = :CARD_IN;
```

Sort and Merge statement

The Sort statement is used to sort records on a set of specified keys and store the sorted records.

The general syntax for sort command is

```
SORT file-name-1
   ON { ASCENDING | DESCENDING } KEY
   data-name-1 [,data-name-2] ...
   USING file-name-2 [,file-name-3] ...
   GIVING file-name-4.
```

Conversion rule for the Sort statement is:

- Declare cursor for each input file of sort command.
- Read the records from each cursor and insert the read record into the table corresponding to the output file.
- If cursor is defined for the table corresponding to the output file specified in the Sort statement, then it is defined with order by clause on the sort condition.

The Merge statement is used to merge one or more sequential files on a set of specified keys and write the merged records in the output file. Conversion rule for the Merge statement is similar to the conversion rule used for the Sort statement.

4.2.10 Unification issue

Unification results in a single integrated view of data. Due to this a record field can be unified to more than one table field. If this happens the file access statements have to be modified appropriately. This modification is necessary because the file access statements are replaced with SQL queries which refer to *sub-fields*.

For example,
File description for CARD_IN is

```
  01 CARD.
     02 ADDRESS PICTURE IS X(80).
```

Unydef for the CARD_IN file is

```
  ADDRESS r {
     CARD_NAME s 25 u,
     CARD_STREET s 25 u,
     CARD_CITY s 30 u,
  }
```

According to the above mentioned conversion rule for the Read statement the following statement is replaced with

```
  READ CARD_IN.

  EXEC SQL fetch CARD_IN into
     :CARD_NAMESqlC1,
     :CARD_STREETSqlC2,
     :CARD_CITYSqlC3
     ;
```

The above SQL query reads data in to *sub-fields* CARD_NAMESSqlC1, CARD_STREETSqlC2, and CARD_CITYSqlC. But the COBOL Read statement reads data in to record field CARD_ADDRESS. This will lead to inconsistency. The above problem is solved by placing statements, which move

data from *sub-fields* to the appropriate record field, after every Read statement. The information regarding the appropriate record field can be obtained from Unydef. For example, we generate the following statements after the above Read statement.

```
sprintf(tempSqlC, "format", CARD_NAMESqlC1,
        CARD_STREETSqlC2, CARD_CITYSqlC3);
sscanf(tempSqlC, "format",CARD_ADDRESS);
```

Similarly, one has to move data in to the appropriate fields before every Write and Rewrite statements.

4.2.11 Procedure division

Procedure division of a COBOL program is made up of paragraphs. Conversion rule for the Procedure division depends on whether paragraph is converted to a function or a block.

The general syntax for the Procedure division of COBOL program is
Procedure division header.
Procedure division body.

Procedure division header is replaced with main or function name depending on whether USING phrase is specified in the Procedure division header or not.

- *Format 1*:

  PROCEDURE DIVISION.
  is replaced with
  main()

- *Format 2*:

  PROCEDURE DIVISION USING data-name-1
  [,data-name-2 *]* ...
  is replaced with
  program_name(data-name-1 *[* ,data-name-2 *]* ...)
  Where program_name is the name of program specified in the Identification division of COBOL program.

The general format for the Procedure division body is

```
program-header
paragraph-name.
    [ sentences ]
    ...
```

Conversion rule for the paragraph when it is converted to a function is:

```
program-header. {
    paragraph-name();
    ...
}
void paragraph name() {
    /sentences /
}
...
```

Conversion rule for the paragraph when it converted to a block is:

```
program-header {
    paragraph-name: {
        /sentences /
        if( PrfrmStmtSqlC())
            goto PrfrmHndlrSqlC;
    }
    ...
}
```

Where PrfrmStmtSqlC checks whether control is transferred to the paragraph by the Perform statement.

Chapter 5

SQLC: A Workbench for Re-engineering

In this chapter we briefly describe the tools provided by SQLC.

5.1 SQLC

SQLC provides an environment to re-engineer existing systems, implemented in COBOL to a relational platform.

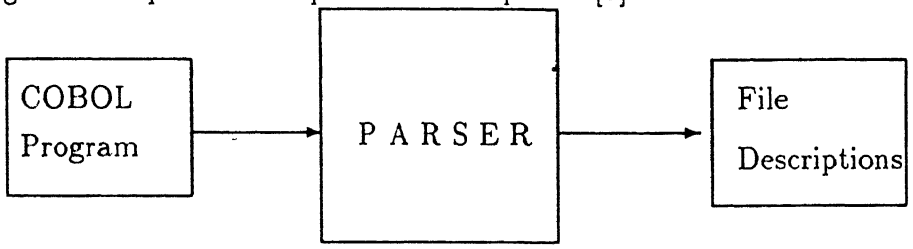
It provides following tools to the user:

- Parser
- Unifyer
- Converter
- Slicer
- Relationer

Parser, Unifyer and Converter are used to migrate COBOL programs to the target environment. Slicer and Relationer displays high level abstractions of the subject system.

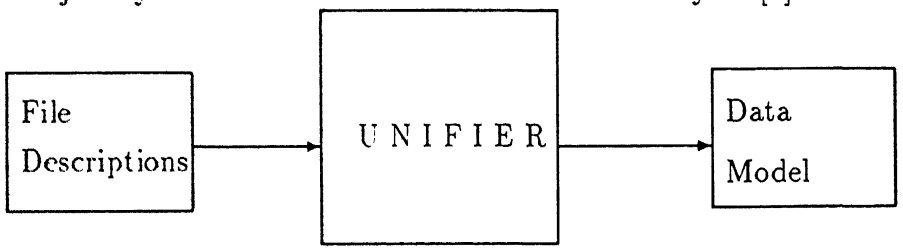
5.1.1 Parser

Parser builds the data view as seen by each program. The data view as seen by each program is described by file descriptions. Parser takes COBOL program as input and outputs file descriptions [3].



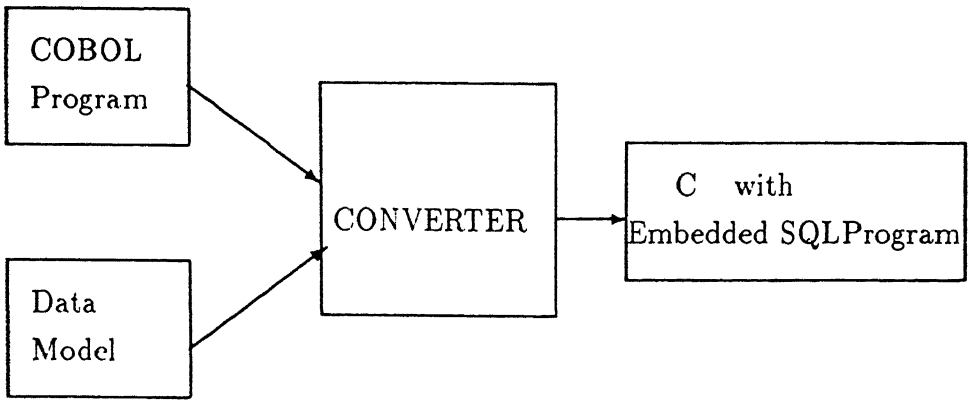
5.1.2 Unifyer

Unifier unifies different data views into a single integrated data view. It takes file descriptions extracted by the parser as input and outputs data model of the subject system in an intermediate form named Unydef [3].



5.1.3 Converter

Converter transforms COBOL program to C with embedded SQL program by applying restructuring rules. It takes the recovered data model and the COBOL program as input and outputs C with embedded.



5.1.4 Slicer

Slicer displays the Program dependence graph (PDG) of the subject system. The PDG specifies the dependencies between the programs of subject system. The main advantage of PDG is reusability. If we are interested in only some programs of subject system, then PDG is helpful in finding out which are all the programs to be re-engineered.

PDG can be constructed with a single pass over the source code. Each node of the graph corresponds to a program or a subroutine. The edges of PDG represent dependencies. There will be an edge from node 'a' corresponding to program 'A' to node 'b' corresponding to program 'B', if program A calls program B. Once the PDG has been constructed, finding out the various programs which are dependent on a given program is straight forward; starting from the node in the graph which corresponds to the program which we are interested in, generate the transitive closure of those nodes by following dependence edges.

5.1.5 Relationer

Relationer provides two features to the user.

1. Display and modify the fields of relations of the subject system.

2. Convert the table to a C file.

Changing the names of fields is to make them more readable. We illustrate a scenario which will result in field names that are not readable. In the

subject system, the data view as seen by each program may be different. One such case is, same data item has different names in different programs. When these fields are unified in Unification, the name of the field is decided randomly. This may lead to names which are not readable. Using this tool user can change the names of the fields of relation to make them more readable.

We are treating all files in COBOL as relations. But this assumption is not always true. For example a file in COBOL can be used to store reports. Such files cannot be treated as tables. By using this tool the user can specify which tables are to be treated as files. This information is used by the converter which generates file access statements instead of embedded SQL statements.

Chapter 6

Characteristics of Re-engineered system

This chapter describes various characteristics of re-engineered system and also outlines the test suite followed to test the restructuring rules.

6.1 Characteristics of Re-engineered system

The re-engineerd system is evaluated on the following characteristics.

- *System life*: The existing COBOL systems have become outdated with current technology. By re-enginering the subject system to a relational platform we are extending the features of a RDBMS environment to it. This will increase the life of the subject system. The re-engineered programs may not be readable as we are using a fully automated procedure for re-engineering the COBOL programs to target environment. But typically the re-engineerd code is considered as a black box and is not maintained further, so the readability is not a issue.

- *Data model*; The data model of the re-engineered system is represented by relations and is not dependent on the application programs. Also, the data view as seen by each program is same.

- *Extensibility*: A Relational platform supports many features like report generator, query language etc., which are not available in conventional

COBOL. These facilities decrease the amount of effort spent by the user in order to extend the features of the subject system. For example, a simple query eliminates the need to write a separate program. Also report generator aids the user in preparing customized reports which is a very tedious process in COBOL. Therefore the re-engineered system is easily extensible.

- *Code length*: The re-engineered code is typically longer than the COBOL program.
  - The COBOL file access statements specify only the name of the file to be accessed. But in C with embedded SQL, in addition to table name corresponding to the file, we have to specify the fields of the table.
  - The COBOL language statements are at a higher level compared to C language statements. As a result some of the COBOL statements are restructured to more than one C statement. For example, the Divide statement in COBOL can specify division for more than one operand which is not possible in C.
  - The language conversion is not straight forward, so the re-engineered code may contain calls to library functions to simulate semantics of COBOL. For example, In COBOL when data is moved to editable data item it gets edited according to the description in its picture class. As C does not provide such a feature, we have to explicitly call a library function to edit the data.
  - The re-engineered object code is larger than COBOL object code as re-engineered object code by default includes code for distributed access, security check and crash recovery.

COBOL program source code (lines)	C with embedded SQL source code (lines)	COBOL program object code (kBytes)	C with embedded SQL object code (kBytes)
43	86	2540	504259
55	172	3308	508406
59	177	3600	508528
78	153	3984	508651
85	188	3872	507419
89	182	3888	507077
105	176	4012	507357
386	1206	12376	535467

- *Execution time*:

COBOL program source code (lines)	C with embedded SQL source code (lines)	COBOL program execution time (secs)	C with embedded SQL execution time (secs)
43	86	1.1	1.5
55	172	1.3	1.9
58	177	1.4	2.3
78	153	1.8	2.4
85	188	2.3	2.6
89	182	2.3	2.8
105	176	2.5	3.1
386	1206	5.2	6.3

A re-engineered program typically takes longer time to execute than a COBOL program. But execution time should not be taken into consideration as the re-engineered program runs on top of a relational platform which provides facilities like security, crash recovery and distributed access. The above table shows execution time for various COBOL program and corresponding C with embedded SQL. The execution time is the average time for 20 runs.

6.2 Test suite

In this section we outline the procedure followed to test the restructuring rules.

For testing the restructuring rules, the COBOL statements can be categorised as Arithmetic and Move statements, Output statements, Sequence control statements and File access statements.

- *Arithmetic and Move statements*: The Arithmetic and Move statements modify the contents of the data items. The conversion rules for these statements were tested by comparing the contents of the data items after execution of the COBOL statement and the corresponding C statement.

- *Output statements*: These statements output the contents of the data items on the console. The conversion rule for these statements were tested by redirecting the the output of both the COBOL program having these statements and the corresponding C with embedded SQL into temporary files and doing a file comparison of these files.

- *Sequence control statements* These statements alter the flow of control. The conversion rules for these statements were tested by compiling the COBOL program which contains these statements and the corresponding C with embedded SQL with debugging option and checking the control flow by tracing these programs using a debugger.

- *File access statements* These statements modify the data stored in the files. The conversion rules for these statements were tested by executing the COBOL program which consists these statements and the corresponding C with embedded SQL and checking whether the data present in the data files and tables is same or not.

Chapter 7

Conclusions

In this thesis, we have discussed the conceptual framework used in re-engineering existing file based COBOL systems to a relational platform and the various restructuring rules used for transforming COBOL programs to C with embedded SQL.

Our main aim has been to develop SQLC, a workbench, which provides an environment to re-engineer information systems which are implemented in COBOL. We have designed and implemented Converter, which transforms the COBOL program into a C with embedded SQL using restructuring rules. We have also constructed the PDG of the subject system which is useful in finding out the reusable components of the subject system. The application of the framework to the existing COBOL systems thus far has demonstrated that it is a practical method for re-engineering. The framework has to be tested against larger systems.

The limitations of Converter are:

- It assumes that COBOL table data items are not present in the file description of a COBOL program.
- It does not handle the variable length records.
- It assumes that all the data items in a record are used in the program.

Future work

- By augmenting data flow analysis in Converter, we can remove the data items that are not used in the program.

- The Goto statements distort the comprehension of the COBOL program. Finding a method which removes Goto statements would be very useful.

- Converter may be integrated in a case tool environment so that it can access the design information of the system. This will increase the functionality of the Converter.

Appendix A

Example program

This appendix presents two COBOL program which have different data views and their equivalent C with embedded SQL programs created by SQLC.

Program 1

```
   IDENTIFICATION DIVISION.
   PROGRAM-ID. READ-RECORDS.
   ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
      SELECT CARD-IN  ASSIGN TO "CARD_RECS".
   DATA DIVISION.
   FILE SECTION.
   FD CARD-IN
      DATA RECORD IS CARD.
   01 CARD.
      02 NAME PICTURE IS X(25).
      02 STREET PICTURE IS X(25).
      02 CITY PICTURE IS X(30).
   WORKING-STORAGE SECTION.
   01 END-OF-DATA-INDICATOR PICTURE IS XXX VALUE IS "NO".
   PROCEDURE DIVISION.
   MAIN-LOGIC.
      OPEN OUTPUT CARD-IN.
      DISPLAY "INSERTING RECORDS INTO FILE CARD-IN"
      PERFORM WRITE-A-CARD
         UNTIL END-OF-DATA-INDICATOR IS EQUAL TO "YES".
```

```
   CLOSE CARD-IN.
   STOP RUN.
WRITE-A-CARD.
   DISPLAY "ENTER NAME:".
   ACCEPT NAME.
   DISPLAY "ENTER STREET:".
   ACCEPT STREET.
   DISPLAY "ENTER CITY:"
   ACCEPT CITY.
   WRITE CARD.
   DISPLAY "ENTER YES TO END READING RECORDS".
   ACCEPT END-OF-DATA-INDICATOR.
```

Program 2

```
   IDENTIFICATION DIVISION.
   PROGRAM-ID. PRINT-RECS.
   ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
      SELECT CARD-IN    ASSIGN TO "CARD-RECS".
   DATA DIVISION.
   FILE SECTION.
   FD CARD-IN
      LABEL RECORDS ARE OMITTED
      DATA RECORD IS CARD.
   01 CARD.
      02 ADDR PICTURE IS X(80).
   WORKING-STORAGE SECTION.
   01 END-OF-DATA-INDICATOR PICTURE IS XXX VALUE IS "NO".
   PROCEDURE DIVISION.
   MAIN-LOGIC.
      OPEN INPUT CARD-IN
      DISPLAY "PRINTING RECORDS FROM FILE CARD-IN"
      PERFORM READ-A-CARD.
      PERFORM PRINT
         UNTIL END-OF-DATA-INDICATOR IS EQUAL TO "YES".
      CLOSE CARD-IN.
      STOP RUN.
   READ-A-CARD.
      READ CARD-IN RECORD AT END
         MOVE "YES" TO END-OF-DATA-INDICATOR.
   PRINT.
      DISPLAY ADDR.
      PERFORM READ-A-CARD.
```

Given below is a table recoverd by SQLC from the above COBOL programs.

```
Table card_recs
(
card_name char(26),
card_street char(26),
card_city char(31)
);
```

Program 1

```
#include <stdio.h>
#include <string.h>
#include "ex.21.h"
#define END_OF_FILE (long)100
#define TRUE  1
#define FALSE  0
#define MAX_REC_SIZE 100
EXEC SQL include sqlca;
char tempSqlC[MAX_REC_SIZE];
char tempSqlC1[MAX_REC_SIZE];
      /* Data Division entries declarations */
      /*Host Variable declarations for file CARD-IN*/
EXEC SQL begin declare section;
   char CARD_NAME[25 + 1];
   char CARD_STREET[25 + 1];
   char CARD_CITY[30 + 1];
EXEC SQL end declare section;

      /*Working Storage entries declarations */
char END_OF_DATA_INDICATOR[3 + 1];

void InitWSEntries() {
    strncpy(END_OF_DATA_INDICATOR,"NO",3);
}

main(){
     InitWSEntries();
     EXEC SQL whenever sqlerror continue;
     fprintf(stdout,"Please wait connecting to INGRES\n");
     EXEC SQL connect  amara ;
       /*Check for connect error*/
     if(sqlca.sqlcode < 0) {
         fprintf(stderr, "Error %ld connecting to database ",
         sqlca.sqlcode );
         exit(1);
     }
     MAIN_LOGIC();
     WRITE_A_CARD();
}

void
MAIN_LOGIC() {
```

```
    fprintf(stdout, "\nINSERTING RECORDS INTO FILE CARD-IN");
    while(!( strcmp(END_OF_DATA_INDICATOR, "YES") == 0 )) {
        WRITE_A_CARD();
    }
    EXEC SQL commit;
    EXEC SQL disconnect;
     exit(0);
}

void
WRITE_A_CARD() {
    fprintf(stdout, "\nENTER NAME:");
    fscanf(stdin,"%25s", CARD_NAME);
    fprintf(stdout, "\nENTER STREET:");
    fscanf(stdin,"%25s", CARD_STREET);
    fprintf(stdout, "\nENTER CITY:");
    fscanf(stdin,"%30s", CARD_CITY);
    EXEC SQL insert into card_recs
        (
        card_name,
        card_street,
        card_city
        )
     values (
        :CARD_NAME,
        :CARD_STREET,
        :CARD_CITY
        );
    fprintf(stdout, "\nENTER YES TO END READING RECORDS");
    fscanf(stdin,"%3s", END_OF_DATA_INDICATOR);
}
```

Program 2

```
---------
# include <stdio.h>
# include <string.h>
# include "ex.22.h"
# define END_OF_FILE (long)100
# define TRUE  1
# define FALSE  0
# define MAX_REC_SIZE 100
EXEC SQL include sqlca;
char tempSqlC[MAX_REC_SIZE];
char tempSqlC1[MAX_REC_SIZE];
    /* Data Division entries declarations */

     /*Host Variable declarations for file CARD-IN*/
EXEC SQL begin declare section;
    char CARD_ADDR[80 + 1];
    char CARD_NAMESqlC1[25 + 1];
    char CARD_STREETSqlC2[25 + 1];
    char CARD_CITYSqlC3[30 + 1];
EXEC SQL end declare section;

         /*Working Storage entries declarations */
char END_OF_DATA_INDICATOR[3 + 1];

void InitWSEntries() {
    strncpy(END_OF_DATA_INDICATOR,"NO",3);
}

main(){
    InitWSEntries();
    EXEC SQL whenever sqlerror continue;
    fprintf(stdout,"Please wait connecting to INGRES\n");
    EXEC SQL connect  amara ;
      /*Check for connect error*/
    if(sqlca.sqlcode < 0) {
        fprintf(stderr, "Error %ld connecting to database ",
        sqlca.sqlcode );
        exit(1);
    }
    MAIN_LOGIC();
    READ_A_CARD();
    PRINT();
}
```

```
void
MAIN_LOGIC() {
      /*Cursor declarations for file CARD-IN*/
    EXEC SQL declare CARD_IN
    cursor for
     select
        card_name ,
        card_street ,
        card_city
    from card_recs;

    EXEC SQL open CARD_IN;
    fprintf(stdout, "\nPRINTING RECORDS FROM FILE CARD-IN");
    READ_A_CARD();
    while(!( strcmp(END_OF_DATA_INDICATOR, "YES") == 0 )) {
        PRINT();
    }
    EXEC SQL close CARD_IN;
    EXEC SQL commit;
    EXEC SQL disconnect;
     exit(0);
}

void
READ_A_CARD() {
    EXEC SQL fetch CARD_IN into
        : CARD_NAMESqlC1 ,
        : CARD_STREETSqlC2 ,
        : CARD_CITYSqlC3
        ;
    sprintf(tempSqlC, "%-25s%-25s%-30s", CARD_NAMESqlC1,
  CARD_STREETSqlC2, CARD_CITYSqlC3);
    strcpy(CARD_ADDR, tempSqlC);
      /*Check for end of table */
    if(sqlca.sqlcode == END_OF_FILE) {
        strncpy(END_OF_DATA_INDICATOR,"YES",3);
    }
}
void
PRINT() {
    fprintf(stdout, "\n%80s",CARD_ADDR);
    READ_A_CARD();
}
```

Appendix B

Example program

This appendix presents a sample COBOL program and its equivalent C with embedded SQL program created by SQLC.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. UPDATE-STUD-DATA.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT CRS-FILE ASSIGN TO "COURSE"
   ORGANIZATION INDEXED
   ACCESS MODE RANDOM
   RECORD KEY IS CRS-ID.
SELECT STUD-FILE ASSIGN TO "STUDENT".
SELECT ENROL-FILE ASSIGN TO "ENROLL".
DATA DIVISION.
FILE SECTION.
FD CRS-FILE.
01 CRS-REC.
   02 CRS-ID    PIC X(6).
   02 CRS-NAME  PIC X(40).
   02 INSTR     PIC X(25).
   02 UNITS     PIC 9.
FD STUD-FILE.
01 STUD-REC.
  02 ROLL-NO        PIC 9(7).
  02 STUD-NAME      PIC X(25).
  02 ADDR           PIC X(30).
  02 UNITS-DONE     PIC 99.
  02 CPI            PIC 9V99.
```

```
FD ENROL-FILE.
01 ENROL-REC.
   02 EN-ROLL-NO  PIC 9(7).
   02 EN-CRS-ID   PIC X(6).
   02 GRADE       PIC X.
WORKING-STORAGE SECTION.
   77  WS-ROLL-NO          PIC 9(7).
   77  WS-CRS-ID           PIC X(6).
   77  WS-UNITS-REGD  PIC 99.
   77  E-O-F          PIC 9 VALUE 0.
   88 STUD-FILE-END   VALUE 1.
   88 CRS-FILE-END    VALUE 2.
   88 ENROL-FILE-END  VALUE 3.
   77  FLAG           PIC 9.
   88 OVER VALUE 1.
PROCEDURE DIVISION.
MAIN-PARA.
   OPEN INPUT STUD-FILE.
   OPEN INPUT CRS-FILE.
   OPEN OUTPUT ENROL-FILE.
   PERFORM  PROCESSING
      UNTIL STUD-FILE-END.
   CLOSE STUD-FILE.
   CLOSE CRS-FILE.
   CLOSE ENROL-FILE.
   STOP RUN.
PROCESSING.
   PERFORM GET-STUDENT.
   IF NOT STUD-FILE-END
      MOVE 0 TO FLAG
   PERFORM REGISTER-COURSES UNTIL OVER.
   DISPLAY WS-UNITS-REGD.
GET-STUDENT.
   READ STUD-FILE RECORD AT END
      MOVE 1 TO E-O-F.
REGISTER-COURSES.
   DISPLAY "CRSNO:".
   ACCEPT WS-CRS-ID.
   IF WS-CRS-ID = '0'
      MOVE 1 TO FLAG
   ELSE
      MOVE WS-CRS-ID TO CRS-ID
      READ CRS-FILE
      COMPUTE WS-UNITS-REGD = WS-UNITS-REGD + UNITS
```

```
MOVE ROLL-NO TO EN-ROLL-NO
MOVE WS-CRS-ID TO EN-CRS-ID
MOVE 'X' TO GRADE
WRITE ENROL-REC.
```

Given below are the tables recoverd by SQLC from the COBOL program.

```
Table course
(
crs_rec_crs_id char(7) not null,
crs_rec_crs_name char(41),
crs_rec_instr char(26),
crs_rec_units integer
);

Table enroll
(
enrol_rec_en_roll_no integer,
enrol_rec_en_crs_id char(7),
enrol_rec_grade char(2)
);

Table student
(
stud_rec_roll_no integer,
stud_rec_stud_name char(26),
stud_rec_addr char(31),
stud_rec_units_done integer,
stud_rec_cpi float
);
```

```
# include <stdio.h>
# include <string.h>
# include "a.cb.h"
# define END_OF_FILE (long)100
# define TRUE  1
# define FALSE  0
# define MAX_REC_SIZE 100
EXEC SQL include sqlca;
char tempSqlC[MAX_REC_SIZE];
char tempSqlC1[MAX_REC_SIZE];
         /*Data Division entries declarations*/
      /*Host Variable declarations for file CRS-FILE*/
EXEC SQL begin declare section;
   char CRS_REC_CRS_ID[6 + 1];
   char CRS_REC_CRS_NAME[40 + 1];
   char CRS_REC_INSTR[25 + 1];
   int CRS_REC_UNITS;
EXEC SQL end declare section;

      /*Host Variable declarations for file STUD-FILE*/
EXEC SQL begin declare section;
   int STUD_REC_ROLL_NO;
   char STUD_REC_STUD_NAME[25 + 1];
   char STUD_REC_ADDR[30 + 1];
   int STUD_REC_UNITS_DONE;
   float STUD_REC_CPI;
EXEC SQL end declare section;

      /*Host Variable declarations for file ENROL-FILE*/
EXEC SQL begin declare section;
    int ENROL_REC_EN_ROLL_NO;
    char ENROL_REC_EN_CRS_ID[6 + 1];
    char ENROL_REC_GRADE[1 + 1];
EXEC SQL end declare section;

         /*Working Storage entries declarations*/
int WS_ROLL_NO;
char WS_CRS_ID[6 + 1];
int WS_UNITS_REGD;
int E_O_F;
int FLAG;

void InitWSEntries() {
    E_O_F = 0;
```

```
}

int OVER() {
    if( FLAG == 1 ) {
        return TRUE;
    }
    return FALSE;
}

int ENROL_FILE_END() {
    if( E_O_F == 3 ) {
        return TRUE;
    }
    return FALSE;
}

int CRS_FILE_END() {
    if( E_O_F == 2 ) {
        return TRUE;
    }
    return FALSE;
}

int STUD_FILE_END() {
    if( E_O_F == 1 ) {
        return TRUE;
    }
    return FALSE;
}

main(){
    InitWSEntries();
    EXEC SQL whenever sqlerror continue;
    fprintf(stdout,"Please wait connecting to INGRES\n");
    EXEC SQL connect  amara ;
      /*Check for connect error*/
    if(sqlca.sqlcode < 0) {
        fprintf(stderr, "Error %ld connecting to database ",
        sqlca.sqlcode );
        exit(1);
    }
    MAIN_PARA();
    PROCESSING();
    GET_STUDENT();
```

```
    REGISTER_COURSES();
}

void
MAIN_PARA() {
      /*Cursor declarations for file STUD-FILE*/
    EXEC SQL declare STUD_FILE
    cursor for
     select
        stud_rec_roll_no,
        stud_rec_stud_name,
        stud_rec_addr,
        stud_rec_units_done,
        stud_rec_cpi
    from student;

    EXEC SQL open STUD_FILE;
    while(!(STUD_FILE_END() )) {
        PROCESSING();
    }
    EXEC SQL close STUD_FILE;
    EXEC SQL commit;
    EXEC SQL disconnect;
     exit(0);
}

void
PROCESSING() {
    GET_STUDENT();
    if(! STUD_FILE_END() ) {
        FLAG = 0;
        while(!(OVER() )) {
            REGISTER_COURSES();
        }
    }
    fprintf(stdout, "\n%2d",WS_UNITS_REGD);
}

void
GET_STUDENT() {
    EXEC SQL fetch STUD_FILE into
        :STUD_REC_ROLL_NO,
        :STUD_REC_STUD_NAME,
        :STUD_REC_ADDR,
```

```
        :STUD_REC_UNITS_DONE,
        :STUD_REC_CPI
        ;
      /*Check for end of table */
    if(sqlca.sqlcode == END_OF_FILE) {
        E_O_F = 1;
    }
}

void
REGISTER_COURSES() {
    fprintf(stdout, "\nCRSNO:");
    fscanf(stdin,"%6s", WS_CRS_ID);
    if( strcmp(WS_CRS_ID, "0") == 0 ) {
        FLAG = 1;
    }
    else {
        strncpy(CRS_REC_CRS_ID,WS_CRS_ID,6);
        EXEC SQL select
            crs_rec_crs_id,
            crs_rec_crs_name,
            crs_rec_instr,
            crs_rec_units
        into
            :CRS_REC_CRS_ID,
            :CRS_REC_CRS_NAME,
            :CRS_REC_INSTR,
            :CRS_REC_UNITS
        from course
         where crs_rec_crs_id = :CRS_REC_CRS_ID;
        WS_UNITS_REGD = WS_UNITS_REGD + CRS_REC_UNITS ;
        AllignSqlC("%2d", &WS_UNITS_REGD);
        ENROL_REC_EN_ROLL_NO = STUD_REC_ROLL_NO;
        strncpy(ENROL_REC_EN_CRS_ID,WS_CRS_ID,6);
        strncpy(ENROL_REC_GRADE,"X",1);
        EXEC SQL insert into enroll
            (
            enrol_rec_en_roll_no,
            enrol_rec_en_crs_id,
            enrol_rec_grade
            )
         values (
            :ENROL_REC_EN_ROLL_NO,
            :ENROL_REC_EN_CRS_ID,
```

```
                :ENROL_REC_GRADE
                );
        }
    }
```

Bibliography

[1] Srinivas, P.L.; *On Re-engineering COBOL programs*, M.Tech Thesis, March 1993, Department of Computer Science, IIT, Kanpur.

[2] *COBCY*, A project, Gnu not unix (GNU), Home page address is: http://rucs2.sunlab.cs.runet.edu/ msharov/cobcy/cobcy.html

[3] Suresh Kumar, S; *Re-engineering COBOL programs*, M.Tech Thesis, 1996, Department of Computer Science, IIT, Kanpur.

[4] Phillippakis, A.S.; Kajmier, L.J.; *Information Systems through COBOL*, McGraw-Hill, 1978.

[5] Roy, M.K; Dastidar D.G.; *COBOL Programming*, Tata McGraw-Hill publishing company limited, 1989.

[6] Application Programming Guide, *HP SQLC*, Hewlett-packard, february, 1988

[7] Aho, A.V.; Sethi, R; and Ullman, J.D.; *Compilers : Principles, Techniques, and Tools*, Reading, Mass.:Addison-Wesley, 1987.

[8] Date, J; *An Introduction to Database systems*, Narosa publishing house, 1990.

[9] Ullman, D.J; *Principles of Database Systems*, Galgotia publication pvt ltd, 1992.

[10] Cross II, J.H.; Chikofsky, E.J.; May Jr., C.H.; *Reverse Engineering* Advances in Computers, Volume 35. Academic Press, Inc. 1992.

[11] Edwards, H.M; Malcom, M; *Recast: Reverse Engineering from COBOL to SSADM specification*, Proceedings in Software Engineering 1993, pages 499-508.

[12] Engberts, A.; Kozaczynski, W.V.; Ning, J.Q.; *Automated Support for Legacy Code Understanding*, Communications of the ACM, May 1994.

[13] Brand, R.; Burson, S.; Kitzmiller, T.; Markosian, L.; Newcomb, P.; *Using an Enabling Technology to Re-engineer Legacy Systems*, Communications of the ACM, May 1994.

[14] Kenny, W.; Scott, R.; Muller, A.h; Storey, M.A.D.; *Structural Redocumentation: A case study* IEEE Software, volume 12, number 1, Jan 1995, pages 46-54.

[15] Bray, O ; Michael, M.H.; *Reengineering a configuration Management System* IEEE Software, volume 12, number 1, Jan 1995, pages 46-54.

[16] S. Subramanyam Sastry.; *Re-engineering COBOL programs*, B.Tech Thesis, April 1995, Department of Computer Science, IIT, Kanpur.